



# Intel® Parallel Studio XE

## Using Intel® Compilers v14.0 and later

***Presented by:**  
Michael D'Mello  
Technical Consulting Engineer  
Software Solutions Group, Intel Corporation  
email: [michael.d'mello@intel.com](mailto:michael.d'mello@intel.com)*

# Agenda

- Why use Intel® C++ Compiler?
- Basic features of the Intel® C++ Compiler
- What's new with the compilers in the 2015 release
- Programming models
- Vectorization
- Offload model
- Summary

# Why use Intel® C++ Compiler?

- Performance

- Vectorization: Processor specific optimizations
- Parallelism: Support for all major paradigms

- Feature Set

- Multiple OS (and Cross Compilation) support
- Integration with major IDE's
- Conformance to standards
- Compatibility with gcc\* and Microsoft\* Compiler
  - For detail see “Compatibility and Portability” sections in:
    - [https://software.intel.com/en-us/compiler\\_14.0\\_ug\\_c](https://software.intel.com/en-us/compiler_14.0_ug_c)
    - [https://software.intel.com/en-us/compiler\\_14.0\\_ug\\_f](https://software.intel.com/en-us/compiler_14.0_ug_f)

- Support

- Premier support: <https://premier.intel.com>
- Forums: <http://software.intel.com/en-us/forum/>

# Common Optimization Switches

Description	Switch
Disable optimization	-O0
Optimize for speed (no code size increase)	-O1
Optimize for speed (default)	-O2
High-level loop optimization	-O3
Create symbols for debugging	-g
Multi-file inter-procedural optimization	-ipo
Profile guided optimization (multi-step build)	-prof-gen -prof-use
Optimize for speed across the entire program **warning: -fast def'n changes over time **don't use this option unless the target is same as host	-fast (same as: -ipo -O3 -no-prec-div -static -xHost)

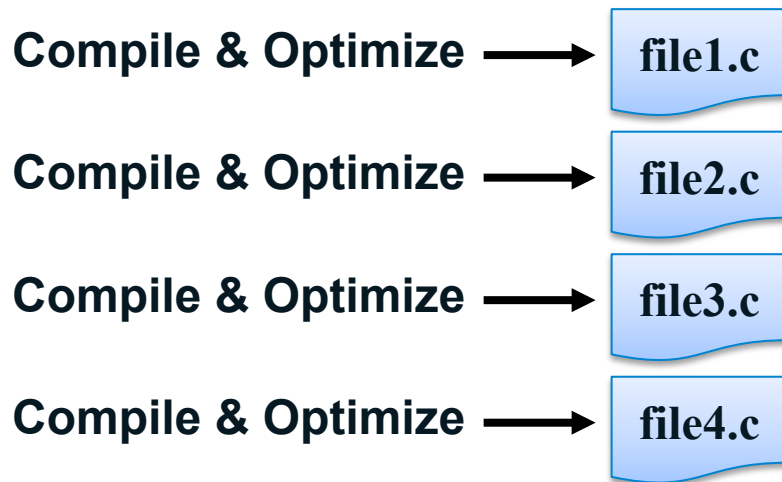
# General optimization options

- -O1
  - ✓ Optimize code size
  - ✓ Auto Vectorization is turned off
- -O2
  - ✓ Inlining
  - ✓ Auto Vectorization
- -O3
  - ✓ Loop optimization
  - ✓ Data pre-fetching
- -ansi-alias / -restrict / -no-prec-div

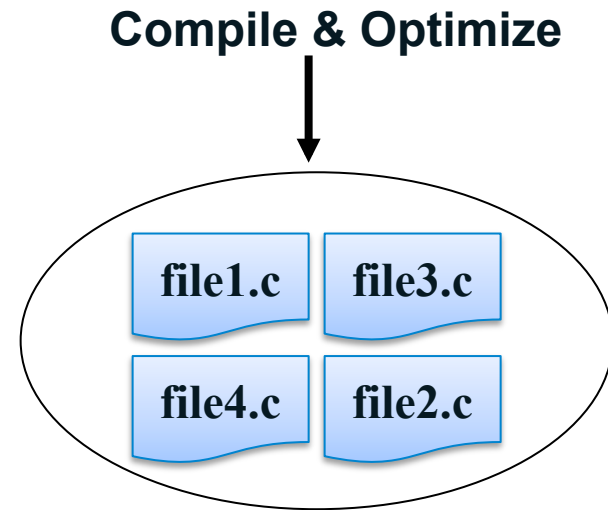
# Interprocedural Optimizations

Extends optimizations across file boundaries

## Without IPO



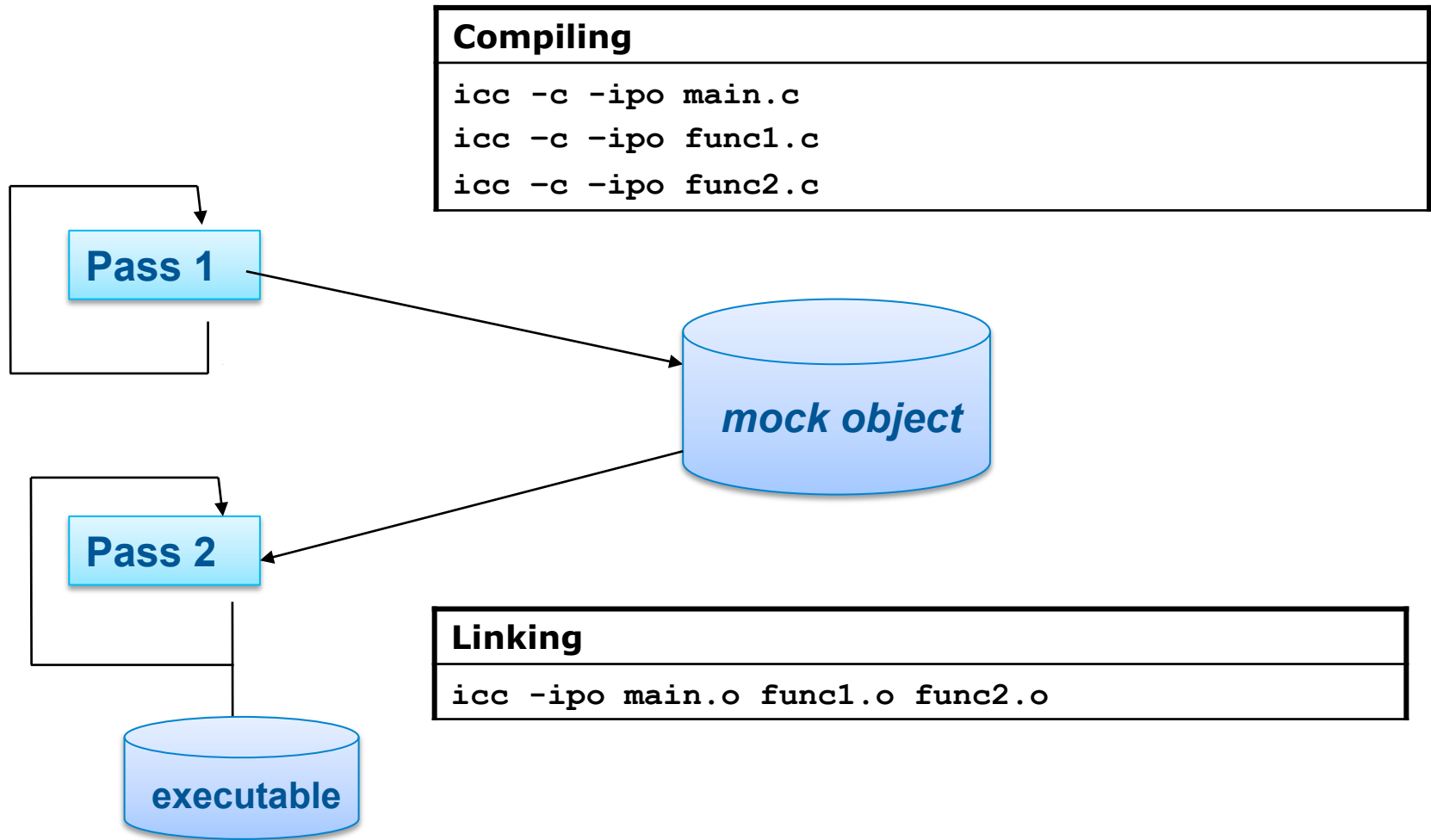
## With IPO



-ip	Only between modules of one source file
-ipo	Modules of multiple files/whole application

# Interprocedural Optimizations (IPO)

Usage: Two-Step Process



# Profile-Guided Optimizations (PGO)

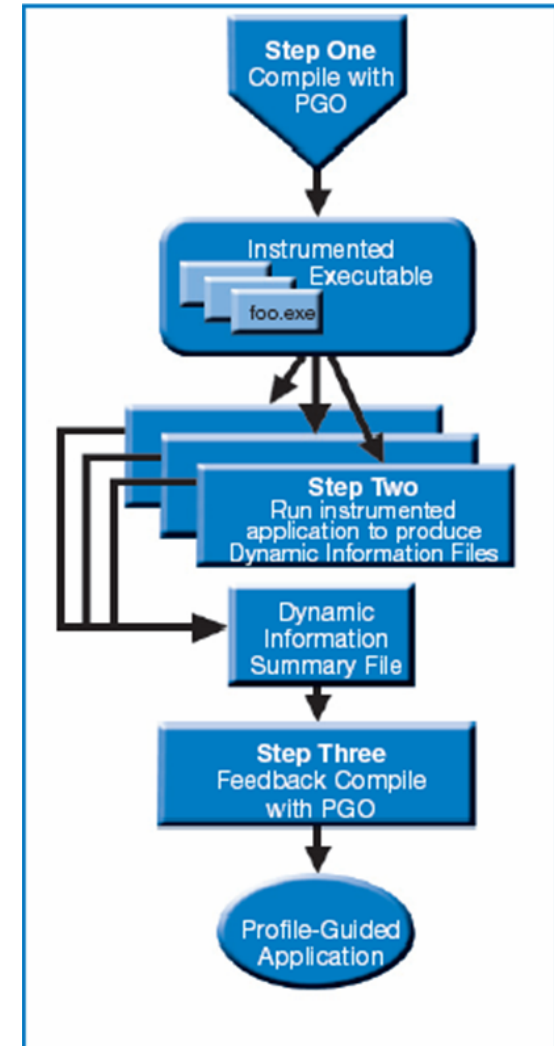
- Static analysis leaves many questions open for the optimizer like:

- How often is  $x > y$
- What is the size of count
- Which code is touched how often

```
if (x > y)
    do_this();
else
    do_that();
```

```
for(i=0; i<count; ++i)
    do_work();
```

- Use execution-time feedback to guide (final) optimization
- Enhancements with PGO:
  - More accurate branch prediction
  - Basic block movement to improve instruction cache behavior
  - Better decision of functions to inline (help IPO)
  - Can optimize function ordering
  - Switch-statement optimization
  - Better vectorization decisions





# PGO Usage: Three Step Process

## Step 1

Compile + link to add instrumentation

```
icc -prof_gen prog.c
```

Instrumented executable:  
prog.exe

## Step 2

Execute instrumented program

prog.exe (on a typical dataset)

Dynamic profile:  
12345678.dyn

## Step 3

Compile + link using feedback

```
icc -prof_use prog.c
```

Merged .dyn files:  
pgopti.dpi

Optimized executable:  
prog.exe

# Guided Automatic Parallelization (GAP)

- Use compiler infrastructure to help developer
  - Vectorization, parallelization and data transformations
  - Extend diagnostic message for failed vectorization and parallelization by specific hints to fix problem
- Exploit multi-year experience brought into the compiler development
  - Performance tuning knowledge based on dealing with numerous applications, benchmarks and compute kernels
- Does not influence code generation
- Use option `-guide/-guide-vec/-guide-data-trans` etc.

# Vectorization Example

```
void f(int n, float *x, float *y, float *z, float *d1, float *d2) {  
    for (int i = 0; i < n; i++)  
        z[i] = x[i] + y[i] - (d1[i]*d2[i]);  
}
```

## GAP Message:

g.c(6): remark #30536: (**LOOP**) Add -no-alias-args option for better type-based disambiguation analysis by the compiler, if appropriate (the option will apply for the entire compilation). This will improve optimizations such as vectorization for the loop at line 6. [**VERIFY**] Make sure that the semantics of this option is obeyed for the entire compilation. [**ALTERNATIVE**] Another way to get the same effect is to add the "restrict" keyword to each pointer-typed formal parameter of the routine "f". This allows optimizations such as vectorization to be applied to the loop at line 6. [**VERIFY**] Make sure that semantics of the "restrict" pointer qualifier is satisfied: in the routine, all data accessed through the pointer must not be accessed through any other

# Data Transformation Example

```
struct S3 {  
    int a;  
    int b; // hot  
    double c[100];  
    struct S2 *s2_ptr;  
    int d;  int e;  
    struct S1 *s1_ptr;  
    char *c_p;  
    int f; // hot  
};
```

```
for (ii = 0; ii < N; ii++){  
    sp->b = ii;  
    sp->f = ii + 1;  
    sp++;  
}
```

peel.c(22): remark #30756: (DTRANS) Splitting the structure 'S3' into two parts will improve data locality and is highly recommended. Frequently accessed fields are 'b, f'; performance may improve by putting these fields into one structure and the remaining fields into another structure. Alternatively, performance may also improve by reordering the fields of the structure. Suggested field order:'b, f, s2\_ptr, s1\_ptr, a, c, d, e, c\_p'. [VERIFY] The suggestion is based on the field references in current compilation ...

# Floating point models

- `fp-model <name>`
  - Method to control the floating point consistency of results
  - **fast**=[1|2] allows more aggressive optimizations at a slight cost in accuracy/consistency (fast=1 is the default)
  - **precise** enables only value safe optimizations
  - **double/extended/source** controls precision of intermediate results
    - double/extended not available in Intel® Fortran
  - For floating point consistency and reproducibility the **recommended** approach is to use **-fp-model precise -fp-model source**

# Intel(r) Libraries

- ICC comes with four optimized libraries

Library	Description
libintlc.so	Intel support libraries
libimf.so	Intel math library
libsvml.so	Short vector math library
libirng.so	Random number generator

- The final binary requires access to these libraries
- Options
  - Include them into OS image
  - Link statically
  - Copy them to the application directory

# Processor Specific Optimizations

- **-x <Target>**
  - Generate specialized code (for Intel® processors) for the specified instruction set
- **-m <Target> (/arch: on Windows)**
  - Generates optimized code that make use of the specified instruction sets
  - Produces binaries that should run on non-Intel processors that implement the capabilities specified by <Target> (for example: SSE3, SSE2, IA32 etc.)
- **-ax <Target>**
  - Generates specialized code path for Intel® processors + default code path
  - Choice of path taken is made by automatic dispatch)
  - Multiple code paths may be specified e.g.: -ax SSE4.1,SSE3
  - Default code path can be modified by additionally using -x or -m

## Note: -xhost

- Optimize for highest instruction set supported by compilation platform

# What's new in Composer XE v15.0 (Beta) (1 of 3)

- New "icl" and "icl++" compilers on OS X\* for improved compatibility with the clang/LLVM\* toolchain
- gcc compatibility features:
  - -ansi-alias enabled by default at -O2 and above on Linux\* C++ to match gcc\* -fstrict-aliasing defaults
  - gcc-compatible function multiversioning
- Full language support for C++ 11
  - -std=c++11, /Qstd=c++11
  - Virtual overrides, Inheriting constructors, Deprecation of exception specifications, User defined literals
- OpenMP 4.0: Except user defined reductions
- Offload (for C, C++) to Intel® Graphics Technology
- Redesigned optimization reports
- Improved lambda function debugging



# What's new in Composer XE v15.0 (Beta) (2 of 3)

- Full Fortran 2003 support
  - parametrized derived types added
- Fortran 2008 Blocks support added
- Nearly complete OpenMP\* 4.0
  - only missing user defined reductions
- Performance improvements in Vectorization
- Fortran option `-init=snan` to initialize all uninitialized SAVEd scalar and array variables of type REAL and COMPLEX to signaling NaNs
- gdb\* debugger supports Fortran
  - Intel debugger removed

# What's new in Composer XE v15.0 (Beta) (3 of 3)

- Ability to create custom install packages from online install
- Compiler option `–no-opt-dynamic-align` to disable generation of multiple code paths depending on alignment of data
- Keyword versions of SIMD pragmas added: `_Simd`, `_Safelen`, `_Reduction`
- Use arithmetical & logical operators with SIMD data types (like `__m128`)  
`__intel_simd_lane()` intrinsic to represent simd lane number in a SIMD vector function
- `-fast/-Ofast` enables `–fp-model fast=2`
- `aligned_new` header
- Permit non-contiguous data transfers on `#pragma offload`

# Intel® Threading Building Blocks

## Generic Parallel Algorithms

Efficient scalable way to exploit the power of multi-core without having to start from scratch

## Concurrent Containers

Common idioms for concurrent access  
- a scalable alternative to a serial container with a lock around it

## TBB Graph

## Task scheduler

The engine that empowers parallel algorithms that employs task-stealing to maximize concurrency

## Thread Local Storage

Scalable implementation of thread-local data that supports infinite number of TLS

## Miscellaneous

Thread-safe timers

## Threads

OS API wrappers

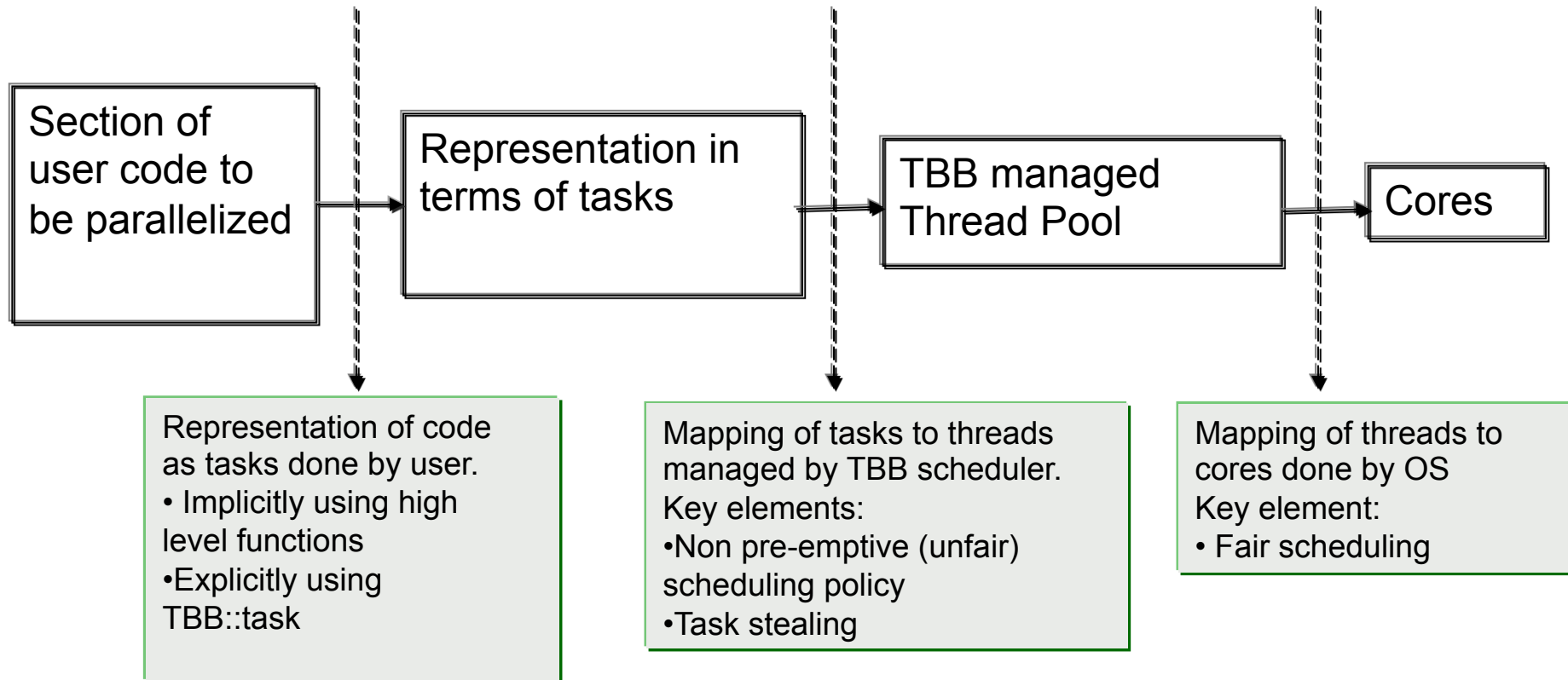
## Synchronization Primitives

User-level and OS wrappers for mutual exclusion, ranging from atomic operations to several flavors of mutexes and condition variables

## Memory Allocation

Per-thread scalable memory manager and false-sharing free allocators

# High level picture of TBB approach to parallelism



# Intel® Cilk™ Plus - Overview

## Task parallelism

### Simple Keywords

Set of keywords, for expression of task parallelism:

```
cilk_spawn  
cilk_sync  
cilk_for
```

### Reducers (Hyper-objects)

Reliable access to nonlocal variables without races

```
cilk::reducer_opadd<int> sum(3);
```

## Data parallelism

### Array Notation

Provide data parallelism for sections of arrays or whole arrays

```
mask[:] = a[:] < b[:] ? -1 : 1;
```

### Elemental Functions

Define actions that can be applied to whole or parts of arrays or scalars

### Execution parameters

Runtime system APIs, Environment variables, pragmas

# Intel® Cilk™ Plus - Overview

- Intel® Cilk™ Plus (Language Extension to C/C++)

## Easier Task & Data Parallelism

3 simple Keywords:

cilk\_for, cilk\_spawn, cilk\_sync

## Intel® Cilk™ Plus Array Notation

Save time with powerful vectorization

Serial code (left) made parallel with Intel Cilk Plus Keywords. No changes to original code.

<pre>int fib (int n) {   if (n &lt;= 2)     return n;   else {     int x,y;     x = fib(n-1);     y = fib(n-2);     return x+y;   } }</pre>	<pre>int fib (int n) {   if (n &lt;= 2)     return n;   else {     int x,y;     x = <b>cilk_spawn</b> fib(n-1);     y = fib(n-2);     <b>cilk_sync</b>;     return x+y;   } }</pre>
---------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Array notation showing simple vector multiplication.

```
a[0:N] = b[0:N] * c[0:N];
```

More sophisticated example of array notation

```
X[0:10:10] = sin(y[20:10:2]);
```

- Multicore Programming with Intel® Cilk™ Plus

**Minimize Software Re-Work for New Hardware**

# Vectorization and Auto-Vectorization

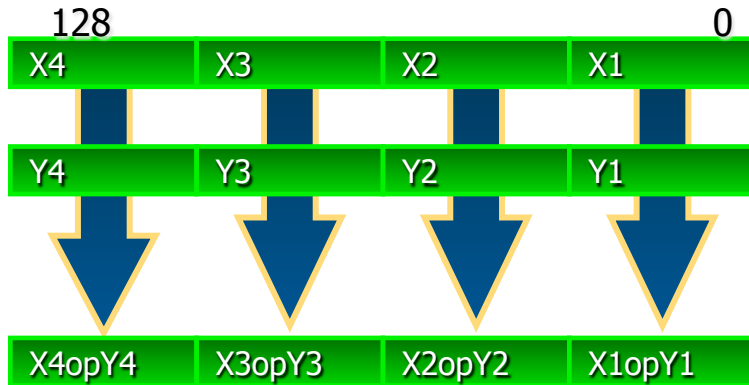
## SIMD – Single Instruction Multiple Data

- Scalar mode
  - one instruction produces one result
- SIMD processing
  - with SSE or AVX instructions
  - one instruction can produce multiple results

```
for (i=0;i<=MAX;i++)  
    c[i]=a[i]+b[i];
```



# Vectorization is Achieved through SIMD Instructions & Hardware



## Intel® SSE

Vector size: 128bit

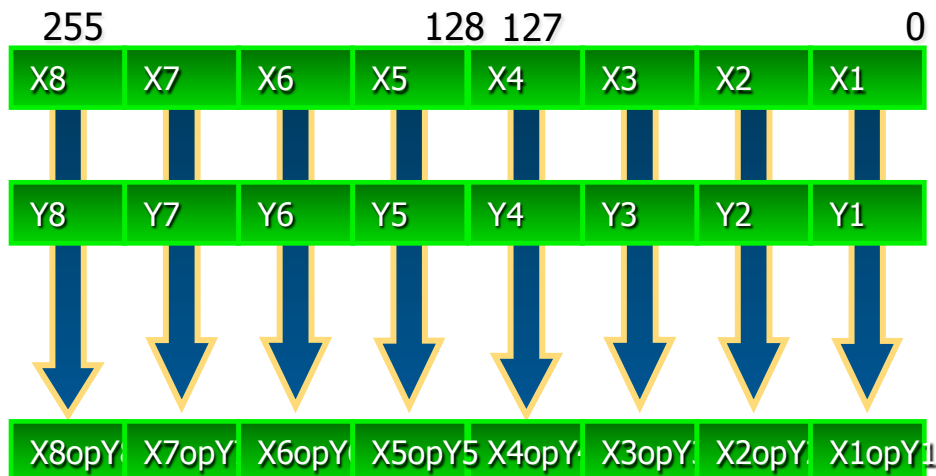
Data types:

8,16,32,64 bit integers

32 and 64bit floats

VL: 2,4,8,16

Sample:  $X_i$ ,  $Y_i$  bit 32 int / float



## Intel® AVX

Vector size: 256bit

Data types: 32 and 64 bit floats

VL: 4, 8, 16

Sample:  $X_i$ ,  $Y_i$  32 bit int or float

First introduced in 2011



# SIMD Data Types for Intel® MIC Architecture

now



16x floats



8x doubles

now



16x 32-bit integers

limited



8x 64-bit integers

Not implemented

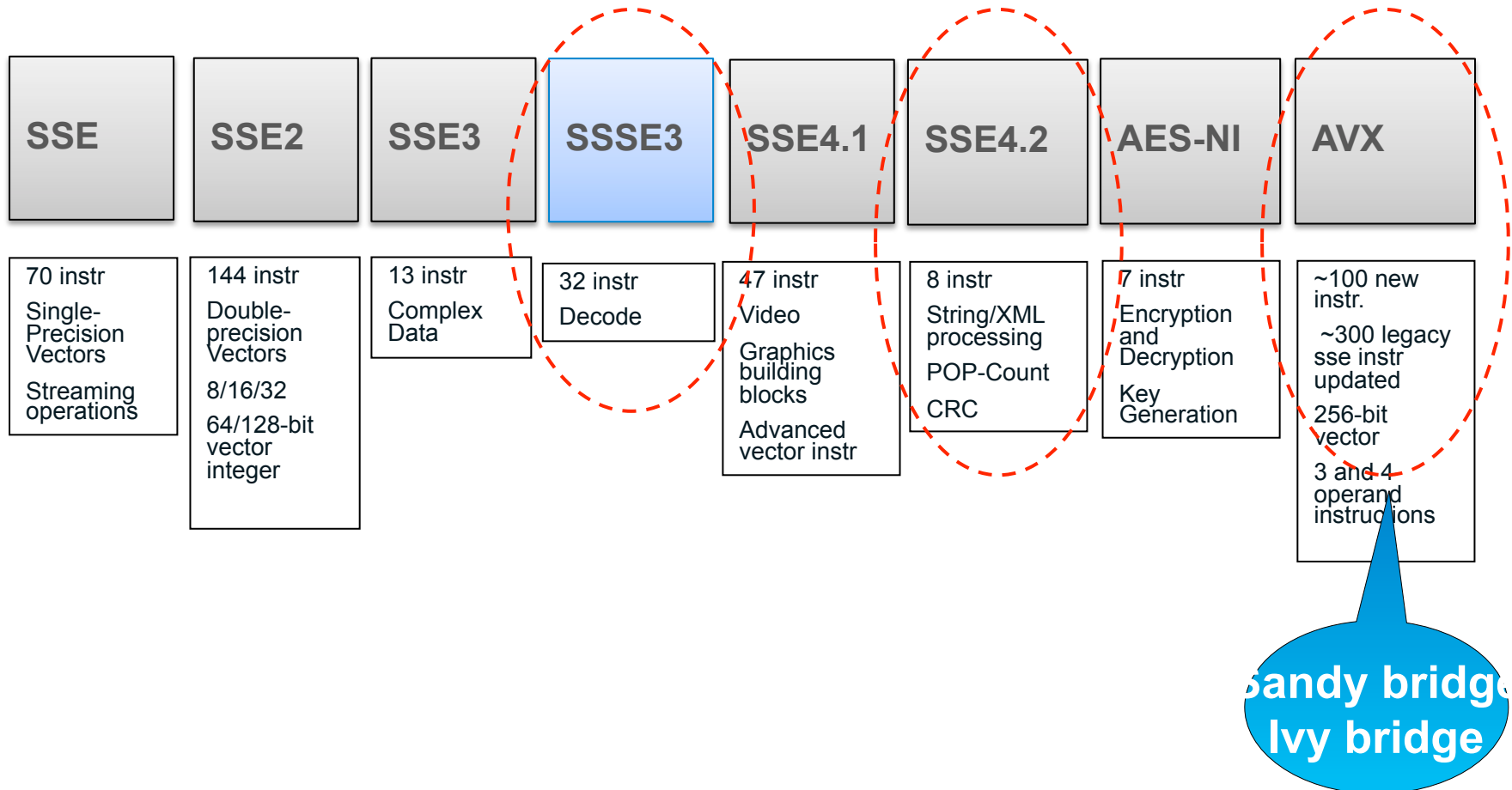


64x 8-bit bytes



32x 16-bit shorts

# SIMD Instruction Enhancements



# Software considerations - Vectorization

**Compiler:**

**Auto-vectorization (no change of code)**

**Compiler:**

**Auto-vectorization hints (`#pragma vector, ...`)**

**Compiler:**

**Intel® Cilk™ Plus Array Notation Extensions**

**SIMD intrinsic class**

**(e.g.: `F32vec`, `F64vec`, ...)**

**Vector intrinsic**

**(e.g.: `_mm_fmadd_pd(...)`, `_mm_add_ps(...)`, ...)**



**Assembler code**

**(e.g.: `[v]addps`, `[v]addss`, ...)**

**Ease of use**

**Programmer control**

# What sorts of loops can be vectorized?

- Countable
  - Loop count must be known at runtime
- Single entry and exit 
- Straight line code 
  - No switch statements for example
- Innermost loop of a nest
- No function calls
  - Exceptions: intrinsics, inlined functions
  - Use `/opt-report-phase ipo_inl` for inlining report

# Vectorization Fine-Control

- Disable vectorization
  - `-no-vec`
  - `#pragma novector`
- Enforce for a single loop if semantically correct
  - `#pragma vector always`
- Ignore vector dependencies
  - `#pragma ivdep`
- “Enforce” vectorization
  - `#pragma simd`
- Generate multiple, feature-specific auto-dispatch code paths for Intel® processors
  - `-ax<code>`

# Compiler Reports (prior to v15.0)

Compiler reporting switches: `-vec-report<n>`, `-par-report<n>`, `openmp-report<n>`, `-opt-report<n>`

Details for `-vec-report<n>`

`n=0`: No diagnostic information

`n=1`: (Default) Loops successfully vectorized

`n=2`: Loops not vectorized – and the reason why not

`n=3`: Adds dependency Information

`n=4`: Reports only non-vectorized loops

`n=5`: Reports only non-vectorized loops and adds dependency info

`n=6`: Reports on vectorized and non-vectorized loops and any proven or assumed data dependences.

`n=7`: reports vectorization summary information and currently requires the use of a Python script to interpret. More information can be found at

<http://software.intel.com/en-us/articles/vecanalysis-python-script-for-annotating-intel-compiler-vectorization-report>

# General note on compiler reports in Intel® Composer XE 2015 (Beta)

- The following is applicable only to Intel® Composer XE 2015 (currently in Beta)
- General
  - Introduced to further enhance the value of reports
  - Applicable to C, C++, and Fortran
  - Applicable to Windows\*, Linux\*, and OS X\*
- Main switches
  - -opt-report=N (Linux\* and OS X\*)
    - N = 1 – 5 for increasing levels of detail (default = 2)
  - -opt-report-phase=str[,str1,str2,...]
    - str=loop, par, vec, openmp, ipo, pgo, cg, offload, tcollect, all
  - -opt-report-file=stdout| stderr | filename

# How to Align Data (C/C++)

Allocate memory on heap aligned to n byte boundary:

```
void* _mm_malloc(int size, int n)
int posix_memalign(void **p, size_t n, size_t size)
```

Alignment for variable declarations:

```
__attribute__((aligned(n))) var_name      or
__declspec(align(n)) var_name
```

## And tell the compiler...

`#pragma vector aligned`

- Asks compiler to vectorize, overriding cost model, and assuming all array data accessed in loop are aligned for targeted processor
- May cause fault if data are not aligned

`__assume_aligned(array, n)`

- Compiler may assume array is aligned to n byte boundary

**n=64 for Intel® Xeon Phi™ coprocessors, n=32 for AVX, n=16 for SSE**



# How to Align Data (Fortran)

Align array on an “n”-byte boundary (n must be a power of 2)

```
!dir$ attributes align:n :: array
```

- Works for dynamic, automatic and static arrays (not in common)

For a 2D array, choose column length to be a multiple of n, so that consecutive columns have the same alignment (pad if necessary)

```
-align array64byte    compiler tries to align all array types
```

## And tell the compiler...

```
!dir$ vector aligned
```

- Asks compiler to vectorize, overriding cost model, and assuming all array data accessed in loop are aligned for targeted processor
- May cause fault if data are not aligned

```
!dir$ assume_aligned array:n    [,array2:n2, ...]
```

- Compiler may assume array is aligned to n byte boundary

**n=64 for Intel® Xeon Phi™ coprocessors, n=32 for AVX, n=16 for SSE**

# Vectorization and Intel® Cilk™ Plus

Vectorization is so important

→ consider explicit vector programming

## Intel® Cilk™ Plus **array notation**

- Compiler can assume LHS does not alias RHS (unlike Fortran)

```
r[n:n] = sqrtf(x[0:n]*x[0:n] + y[0:n]*y[0:n]);
```

## and **simd-enabled functions**

- Allow vectorization over function calls, without inlining

```
__attribute__((vector)) float myfun(float a, float x, float y)
{...}
```

```
z[:] = myfun1(a[:], b[:], c[:]);
```

## **#pragma simd**

- Directs compiler to vectorize if at all possible
- Overrides all dependencies and cost model
  - More aggressive than pragmas ivdep and vector always
- Semantics modeled on OpenMP parallel pragmas
  - Private and reduction clauses **required** for correctness

# Explicit Vector Programming:

## Intel® Cilk™ Plus Array notation

```
void addit(double* a, double* b,  
int m, int n, int x)  
{  
    for (int i = m; i < m+n; i++) {  
        a[i] = b[i] + a[i-x];  
    }  
}
```

loop was not vectorized:  
existence of vector dependence.

```
void addit(double* a, double * b,  
int m, int n, int x)  
{  
    // I know x<0  
    a[m:n] = b[m:n] + a[m-x:n];  
}
```

LOOP WAS VECTORIZED.

Array notation asks the compiler to vectorize

- asserts this is safe (for example,  $x < 0$ )
- Improves readability

# Explicit Vector Programming:

## Intel® Cilk™ Plus pragma example

Using `#pragma simd` (C/C++) or `!DIR$ SIMD` (Fortran)  
or `#pragma omp simd` (OpenMP 4.0)

```
void addit(double* a, double* b,
int m, int n, int x)
{
    for (int i = m; i < m+n; i++) {
        a[i] = b[i] + a[i-x];
    }
}
```

loop was not vectorized:  
existence of vector dependence.

```
void addit(double* a, double * b,
int m, int n, int x)
{
    #pragma simd      // I know x<0
    for (int i = m; i < m+n; i++) {
        a[i] = b[i] + a[i-x];
    }
}
```

SIMD LOOP WAS VECTORIZED.

- Use when you **KNOW** that a given loop is safe to vectorize
- The Intel Compiler will vectorize if at all possible  
(will ignore dependency or efficiency considerations)
- Minimizes source code changes needed to enforce vectorization

# SIMD-enabled Function

Compiler generates vector version of a scalar function that can be called from a vectorized loop:

```
__attribute__((vector(uniform(y, xp, yp))))  
float func(float x, float y, float xp, float yp) {  
    float denom = (x-xp)*(x-xp) + (y-yp)*(y-yp);  
    denom= 1./sqrtf(denom);  
    return denom;  
}
```

y, xp and yp are constant,  
x can be a vector

func\_vec.f(1): (col. 21) remark: FUNCTION WAS VECTORIZED.

```
#pragma simd private(x) reduction(+:sumx)
```

```
    for (i=1; i<nx; i++) {  
        x = x0 + (float)i * h;  
        sumx = sumx + func(x, y ,xp, yp);  
    }  
    enddo
```

These clauses are  
required for correctness,  
just like for OpenMP

SIMD LOOP WAS VECTORIZED.

# Prefetching - automatic

Compiler prefetching is on by default for the Intel® Xeon Phi™ coprocessor at -O2 and above

- Prefetches issued for regular memory accesses inside loops
- But not for indirect accesses `a[index[i]]`
- More important for Intel Xeon Phi coprocessor (in-order)  
than for Intel® Xeon® processors (out-of-order)
- Very important for apps with many L2 cache misses

Use the compiler reporting options to see detailed diagnostics of prefetching per loop

`-opt-report-phase hlo -opt-report 3` e.g.

Total #of lines prefetched in main for loop at line 49=4

Using noloc distance 8 for prefetching unconditional memory reference in stmt at line 49

Using second-level distance 2 for prefetching spatial memory reference in stmt at line 50

`-opt-prefetch=n` (4 = most aggressive) to control

`-opt-prefetch=0` or `-no-opt-prefetch` to disable

`-opt-prefetch-distance =<n1> [,<n2>]` to tune how far ahead to prefetch

# Prefetching - manual

## Use intrinsics

```
_mm_prefetch((char *) &a[i], hint);
```

See `xmmintrin.h` for possible hints (for L1, L2, non-temporal, ...)

```
MM_PREFETCH(A, hint)    for Fortran
```

- But you have to figure out and code how far ahead to prefetch
- Also gather/scatter prefetch intrinsics, see `zmmmintrin.h` and compiler user guide, e.g. `_mm512_prefetch_i32gather_ps`

## Use a pragma / directive (easier):

```
#pragma prefetch  a    [:hint[:distance]]
```

```
!DIR$ PREFETCH A, B, ...
```

- You specify what to prefetch, but can choose to let compiler figure out how far ahead to do it.

## Hardware L2 prefetcher is also enabled by default

- If software prefetches are doing a good job, then hardware prefetching does not kick in

# Streaming Stores

Write directly to memory bypassing cache

- for “nontemporal” data that are not read and will not be reused
- avoids “read for ownership” to get line into cache
  - Reduces memory bandwidth requirements
  - Keeps cache available for useful work, avoids “pollution”

`#pragma vector nontemporal (v1, v2, ...)` hint to compiler

- No Streaming Stores:  
448 Bytes read/write per iteration
- With Streaming Stores:  
320 Bytes read/write per iteration
- `-vec-report6` shows  
what the compiler did

```
for (int chunk = 0; chunk < OptPerThread; chunk += CHSIZE)
{
    #pragma simd vectorlength(CHSIZE)
    #pragma vector aligned
    #pragma vector nontemporal (CallResult, PutResult)
    for(int opt = chunk; opt < (chunk+CHSIZE); opt++)
    {
        float T = OptionYears[opt];
        float X = OptionStrike[opt];
        float S = StockPrice[opt];

        .....
        CallVal = S * CNDD1 - XexpRT * CNDD2;
        PutVal = CallVal + XexpRT - S;
        CallResult[opt] = CallVal ;
        PutResult[opt] = PutVal ;
    }
}
```

bs\_test\_sp.c(215): (col. 4) remark: vectorization support: streaming store was generated for CallResult.  
bs\_test\_sp.c(216): (col. 4) remark: vectorization support: streaming store was generated for PutResult.



# OpenMP on the Coprocessor

The basics work just like on the host CPU

- For both native and offload models
- Need to specify `-openmp`

There are 4 hardware thread contexts per core

- Need at least 2 x ncore threads for good performance
  - For all except the most memory-bound workloads
  - Often, 3x or 4x (number of available cores) is best
  - Very different from hyperthreading on the host!
  - `-opt-threads-per-core=n` advises compiler how many threads to optimize for
- If you don't saturate all available threads, be sure to set **KMP\_AFFINITY** to control thread distribution

# OpenMP defaults

`OMP_NUM_THREADS` defaults to

- 1 x ncore for host (or 2x if hyperthreading enabled)
- 4 x ncore for native coprocessor applications
- 4 x (ncore-1) for offload applications
  - one core is reserved for offload daemons and OS (typically the highest numbered)
- Defaults may be changed via environment variables or via API calls on either the host or the coprocessor

# Target OpenMP environment (offload)

Use target-specific APIs to set for coprocessor target only, e.g.

`omp_set_num_threads_target()` (called from host)

`omp_set_nested_target()` etc.

- Protect with `#ifdef __INTEL_OFFLOAD`, undefined with `-no-offload`
- Fortran: `USE MIC_LIB` and `OMP_LIB` C: `#include <offload.h>`

Or define coprocessor – specific versions of env variables using

`MIC_ENV_PREFIX=PHI` (no underscore)

- Values on coprocessor **no longer default to values on host**
- Set values specific to coprocessor using

`export PHI_OMP_NUM_THREADS=120` (all coprocessors)

`export PHI_2_OMP_NUM_THREADS=180` for coprocessor #2, etc.

`export PHI_3_ENV="OMP_NUM_THREADS=240|KMP_AFFINITY=balanced"`

# Thread Affinity Interface

Allows OpenMP threads to be bound to physical or logical cores

- export environment variable `KMP_AFFINITY=`
  - `physical` use all physical cores before assigning threads to other logical cores (other hardware thread contexts)
  - `compact` assign threads to consecutive h/w contexts on same physical core (e.g. to benefit from shared cache)
  - `scatter` assign consecutive threads to different physical cores (eg to maximize access to memory)
  - `balanced` blend of compact & scatter (currently only available for Intel® MIC Architecture)
- Helps optimize access to memory or cache
- Particularly important if all available h/w threads not used
  - else some physical cores may be idle while others run multiple threads
- See compiler documentation for (much) more detail

# Example – share work between coprocessor and host using OpenMP

```
omp_set_nested(1);
#pragma omp parallel private(ip)
{
    #pragma omp sections
    {
        #pragma omp section
        /*      use pointer to copy back only part of potential array,
           to avoid overwriting host */
        #pragma offload target(mic) in(xp) in(yp) in(zp) out(ppot:length(np1))
        #pragma omp parallel for private(ip)
            for (i=0;i<np1;i++) {
                ppot[i] = threed_int(x0,xn,y0,yn,z0,zn,nx,ny,nz,xp[i],yp[i],zp[i]);
            }
        #pragma omp section
        #pragma omp parallel for private(ip)
            for (i=0;i<np2;i++) {
                pot[i+np1] =
                threed_int(x0,xn,y0,yn,z0,zn,nx,ny,nz,xp[i+np1],yp[i+np1],zp[i+np1]);
            }
    }
}
```

Top level, runs on host  
Runs on coprocessor  
Runs on host

# Useful Links

More information on Intel's software offerings and services at <http://software.intel.com>

Support: <http://premier.intel.com>

C++ Compiler Basics:

<https://software.intel.com/en-us/c-compilers#pid-16692-920>

Compilation for Intel® Xeon PHI Coprocessor:

<https://software.intel.com/en-us/articles/compilation-for-intel-xeon-phi-coprocessor-vectorization-alignment-pre-fetch-more/>

Intel® Software Development Tools 2015 Beta:

<https://software.intel.com/en-us/articles/intel-software-development-tools-2015-beta> (a number of useful videos are available)

# Some more useful links

- For tutorials on various aspects of vectorization including SIMD pragmas
  - <https://software.intel.com/articles/vectorization-essentials>
- For information on asynchronous computation via the offload model:
  1. <https://software.intel.com/en-us/videos/advanced-intel-xeon-phi-coprocessor-workshop-memory-part-1-basics>
  2. <https://software.intel.com/en-us/videos/advanced-intel-xeon-phi-coprocessor-workshop-memory-part-2-performance-tuning>
- For a glimpse of a slightly different use case of the offload model:
  - <https://software.intel.com/en-us/videos/intel-c-compiler-unveils-compute-power-of-intel-graphics-technology-for-general-purpose>

# Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

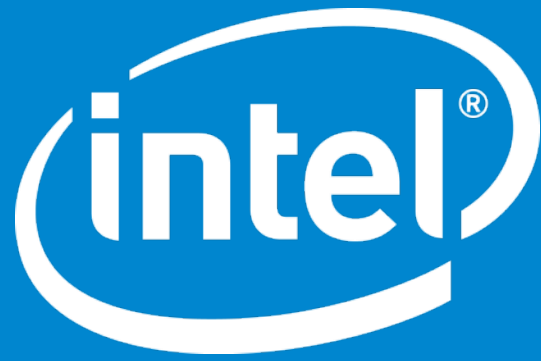
Copyright © 2014, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

## Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804





# Single entry and exit

```
void no_vec(float a[], float b[], float c[])  
{  
    int i = 0.;  
    while (i < 100) {  
        a[i] = b[i] * c[i];  
        // this is a data-dependent exit condition:  
        if (a[i] < 0.0) break;  
        ++i ;  
    }  
}
```

```
> icc -c -O2 -vec-report2 two_exits.cpp
```

**two\_exits.cpp(4) (col. 9): remark: loop was not vectorized:**

**nonstandard loop is not a vectorization candidate.**

# Straight Line Code

```
#include <math.h>

void quad(int length, float *a, float *b,
float *c, float *restrict x1, float *restrict x2)
{
    for (int i=0; i<length; i++) {
        float s = b[i]*b[i] - 4*a[i]*c[i];
        if ( s >= 0 ) { s = sqrt(s) ; x2[i] = (-b[i]+s)/(2.*a[i]); x1[i] = (-b[i]-s)/(2.*a[i]);
        }
        else { x2[i] = 0.; x1[i] = 0.;
            }
    }
}
```

```
> icc -c -restrict -vec-report2 quad.cpp
```

```
quad5.cpp(5) (col. 3): remark: LOOP WAS VECTORIZED.
```